CSC 311 – Winter 2022-2023

Design and Analysis of Algorithms
6. Divide-and-Conquer

Prof. Mohamed Menai
Department of Computer Science
King Saud University

Outline

- Divide-and-Conquer
- Binary search
- Merge sort
- Quicksort
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Quickhull algorithm

Divide-and-Conquer

- Breaking large problems into smaller subproblem instances

The most-well known algorithm design strategy:

1. Divide the instance of problem into two or more smaller instances (subproblems).
2. Conquer the smaller instances by solving them recursively.
3. Combine the solutions to the smaller instances into the solution for the original (larger) instance.

Divide-and-Conquer

- Given: a divide-and-conquer algorithm
  - An algorithm that divides the problem of size n into a subproblems, each of size n/b
  - Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function $f(n)$
- $T(n) = aT(n/b) + f(n)$

4

Binary search

Very efficient algorithm for searching in sorted array:

K

vs

A[0] . . . A[m] . . . A[n-1]

If K = A[m], stop (successful search); otherwise, continue searching by the same method in A[0..m-1] if K < A[m] and in A[m+1..n-1] if K > A[m]

```
l ← 0;   r ← n-1
while l ≤ r do
    m ← ⌊(l+r)/2⌋
     if  K = A[m]  return m
     else if K < A[m]  r ← m-1
     else l ← m+1
return -1
```

Analysis of binary search

- Time efficiency
  - Recurrence: $T(n) = 1 + T(\lfloor n/2 \rfloor)$, $T(1) = 1$
  - Solution: $T(n) = \lceil \log(n+1) \rceil = \Theta(\log n)$

    This is VERY fast: e.g., $T(10^6) = 20$
- Optimal for searching a sorted array
- Limitations: must be a sorted array (not linked list)

6

Merge sort

```
MergeSort(A, left, right)
    if (left < right)
        mid = floor((left + right) / 2);
        MergeSort(A, left, mid);
        MergeSort(A, mid+1, right);
        Merge(A, left, mid, right);

// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A
// (how long should this take?)
```

Analysis of merge sort

Statement	Effort
MergeSort(A, left, right)	T(n)
if (left < right)	Θ(1)
mid = floor((left + right) / 2);	Θ(1)
MergeSort(A, left, mid);	T(n/2)
MergeSort(A, mid+1, right);	T(n/2)
Merge(A, left, mid, right);	Θ(n)

- So $T(n) = \Theta(1)$ when $n = 1$, and

  $= 2T(n/2) + n$ when $n > 1$

  $T(n) = ?$

Quicksort

- Sorts in place
- Sorts $O(n \log n)$ in the average case
- Sorts $O(n^2)$ in the worst case
  - But in practice, it's quick
  - And the worst case doesn't happen often

Quicksort

- Another divide-and-conquer algorithm
  - The array A[p..r] is partitioned into two non-empty subarrays A[p..q] and A[q+1..r]
    - Invariant: All elements in A[p..q] are less than all elements in A[q+1..r]
  - The subarrays are recursively sorted by calls to quicksort
  - Unlike merge sort, no combining step: two subarrays form an already-sorted array

Quicksort code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q);
        Quicksort(A, q+1, r);
    }
}
```

Partition

- All the action takes place in the **`partition()`** function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray ≤ all values in second subarray
  - Returns the index of the "pivot" element separating the two subarrays

Partition

- Partition(A, p, r):
  - Select an element to act as the “pivot”
  - Grow two regions, A[p..i] and A[j..r]
    - All elements in A[p..i] <= pivot
    - All elements in A[j..r] >= pivot
  - Increment i until A[i] >= pivot
  - Decrement j until A[j] <= pivot
  - Swap A[i] and A[j]
  - Repeat until i >= j
  - Return j

13

Partition code

```
Partition(A, p, r)
    x = A[p];
    i = p - 1;
    j = r + 1;
    while (TRUE)
        repeat
            j--;
        until A[j] <= x;
        repeat
            i++;
        until A[i] >= x;
        if (i < j)
            Swap(A, i, j);
        else
            return j;
```

`Illustrate on`
`A = {5, 3, 2, 6, 4, 1, 3, 7};`

What is the running time of **`partition()`**?

Partition code

```
Partition(A, p, r)
    x = A[p];
    i = p - 1;
    j = r + 1;
    while (TRUE)
        repeat
            j--;
        until A[j] <= x;
        repeat
            i++;
        until A[i] >= x;
        if (i < j)
            Swap(A, i, j);
        else
            return j;
```

`partition()` runs in $O(n)$ time

Analyzing quicksort

- Worst case for the quicksort
  - Partition is always unbalanced
- Best case for the quicksort
  - Partition is perfectly balanced
- Which is more likely?
  - The latter...
- Will any particular input elicit the worst case?
  - Yes: Already-sorted input

Analyzing quicksort

- In the worst case:

  $T(1) = \Theta(1)$

  $T(n) = T(n - 1) + \Theta(n)$

- Works out to

  $T(n) = \Theta(n^2)$

17

Analyzing quicksort

- In the best case:

  $T(n) = 2T(n/2) + \Theta(n)$
- Works out to

  $T(n) = \Theta(n \log n)$

Improving quicksort

- The real liability of quicksort is that it runs in $O(n^2)$ on already sorted input
- Two solutions:
  - Randomize the input array, OR
  - Pick a random pivot element
- How will these solve the problem?
  - By insuring that no particular input can be chosen to make quicksort run in $O(n^2)$ time

Multiplication of large integers

Consider the problem of multiplying two (large) n-digit integers represented by arrays of their digits such as:

A = 12345678901357986429   B = 87654321284820912836

The grade-school algorithm:

$$
\begin{array}{r}
a_1\ a_2 \dots\ a_n \\
\underline{b_1\ b_2 \dots\ b_n} \\
(d_{10})\ d_{11} d_{12} \dots\ d_{1n} \\
(d_{20})\ d_{21} d_{22} \dots\ d_{2n} \qquad \\
\dots\ \dots\ \dots\ \dots\ \dots\ \dots\ \dots \\
\underline{(d_{n0})\ d_{n1} d_{n2} \dots\ d_{nn}} \qquad\qquad
\end{array}
$$

Efficiency: $\Theta(n^2)$ one-digit multiplications

First Divide-and-Conquer algorithm

A small example: $A * B$ where $A = 2135$ and $B = 4014$

$A = (21 \cdot 10^2 + 35)$, $B = (40 \cdot 10^2 + 14)$

So, $A * B = (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$

$= 21 * 40 \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14$

In general, if $A = A_1A_2$ and $B = B_1B_2$ (where A and B are *n*-digit,

A_1, A_2, B_1, B_2 are $n/2$-digit numbers),

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

Recurrence for the number of one-digit multiplications $M(n)$:

$$M(n) = 4M(n/2), \quad M(1) = 1$$

Solution: $M(n) = n^2 = \Theta(n^2)$

21

Second Divide-and-Conquer algorithm

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

The idea is to decrease the number of multiplications from 4 to 3:

$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,$

i.e., $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2,$

which requires only 3 multiplications at the expense of (4-1) extra add/sub. Recurrence for the number of multiplications M(n):

$$M(n) = 3M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585} = \Theta(n^{1.585})$

22

Strassen's matrix multiplication

Strassen observed [1969] that the product of two matrices can be computed as follows:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

23

Formulas for Strassen's algorithm

$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$

$M_2 = (A_{10} + A_{11}) * B_{00}$

$M_3 = A_{00} * (B_{01} - B_{11})$

$M_4 = A_{11} * (B_{10} - B_{00})$

$M_5 = (A_{00} + A_{01}) * B_{11}$

$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$

$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$

Analysis of Strassen's algorithm

- If n is not a power of 2, matrices can be padded with zeros.
- Number of multiplications:

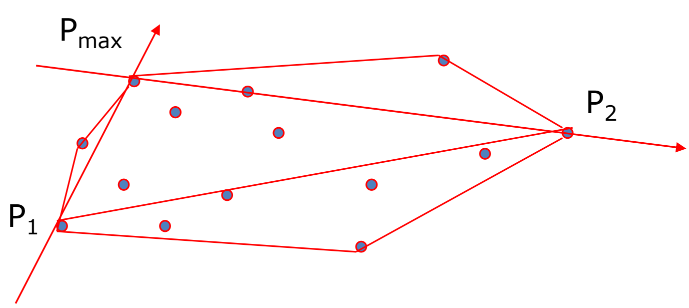
$$M(n) = 7M(n/2), \quad M(1) = 1$$

- Solution: $M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$ vs. n^3 of brute-force alg. $M(n) = \Theta(n^{2.807})$
- Algorithms with better asymptotic efficiency are known but they are even more complex.

Quickhull algorithm

Convex hull: smallest convex set that includes given points

- Assume points are sorted by *x*-coordinate values
- Identify *extreme points* P_1 and P_2 (leftmost and rightmost)
- Compute *upper hull* recursively:
  - find point P_{max} that is farthest away from line P_1P_2
  - compute the upper hull of the points to the left of line P_1P_{max}
  - compute the upper hull of the points to the left of line $P_{max}P_2$
- Compute *lower hull* in a similar manner



Efficiency of quickhull algorithm

- Finding point farthest away from line P_1P_2 can be done in linear time
- Time efficiency:
  - worst case: $\Theta(n^2)$ (as quicksort)
  - average case: $\Theta(n)$ (under reasonable assumptions about distribution of points given)
- If points are not initially sorted by x-coordinate value, this can be accomplished in $O(n \log n)$ time
- Several $O(n \log n)$ algorithms for convex hull are known

Reading

Chapter 4

Anany Levitin, Introduction to the design and analysis of algorithms, 3rd Edition, Pearson, 2011.